Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics

Antoine Miné
École Normale Supérieure, Paris, France
mine@di.ens.fr

Abstract

We propose a memory abstraction able to lift existing numerical static analyses to C programs containing union types, pointer casts, and arbitrary pointer arithmetics. Our framework is that of a combined points-to and data-value analysis. We abstract the contents of compound variables in a field-sensitive way, whether these fields contain numeric or pointer values, and use stock numerical abstract domains to find an overapproximation of all possible memory states—with the ability to discover relationships between variables. A main novelty of our approach is the dynamic mapping scheme we use to associate a flat collection of abstract cells of scalar type to the set of accessed memory locations, while taking care of byte-level aliases—*i.e.,* C variables with incompatible types allocated in overlapping memory locations. We do not rely on static type information which can be misleading in C programs as it does not account for all the uses a memory zone may be put to.

Our work was incorporated within the Astrée static analyzer that checks for the absence of run-time-errors in embedded, safety-critical, numerical-intensive software. It replaces the former memory domain limited to well-typed, union-free, pointer-cast free data-structures. Early results demonstrate that this abstraction allows analyzing a larger class of C programs, without much cost overhead.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Assertion checkers, Formal methods, Validation; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Assertions, Invariants, Mechanical verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Reliability, Experimentation, Languages, Theory, Verification

Keywords Abstract Interpretation, Points-to Analysis, Numerical Analysis, Critical Software

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.

1. Introduction

In embedded critical software, the slightest programming error can have the most disastrous consequences. Even when the high-level specification of a software is correct, its actual implementation using efficient but unsafe low-level languages can introduce new kinds of bugs, such as run-time errors triggered by integer wrap-around or invalid floating-point operations—witness the demise of the Ariane launcher in 1996 [8]. Hence, there is a demand for tools able to check for potential run-time errors in low-level programs, in an automatic and sound way. To this end, we focus here on deriving the set of values the variables of a C program can take during all its executions. We allow sound but generally incomplete approximations to ensure an efficient analysis. This way, we are able to report a set of alarms that encompasses *all* possible run-time error situations. Hopefully, when the analysis is sufficiently precise, there are zero alarms which actually *proves formally* the absence of run-time errors.

Unfortunately, the weak type system of the C programming language complicates value analysis greatly. In the presence of union types, pointer arithmetics or pointer casts, the same sequence of memory bytes can be manipulated as values of distinct types. Most existing analyses avoid the problem by either restricting the input language or by being overly conservative about the contents of memory locations that can be accessed with incompatible types—*i.e.*, treat them in a field-insensitive way. We found these solutions to be insufficient to analyze actual embedded C codes provided by industrial end-users. As they exploit some knowledge of the bit-representation of values and the low-level semantics of operators, tracking precisely the manipulated values is required to prove the absence of run-time errors.

To address these problems, we propose a field-sensitive value analysis for C programs containing union types, pointer arithmetics and pointer casts. Our main contribution is an abstraction that maps the memory, viewed as untyped spans of bytes, to a collection of synthetic cells with integer or floating-point type. We then rely on existing alias-unaware numerical analyses—such as intervals [5] or octagons [15]—to infer numerical invariants on cells. Our abstraction translates operations on byte-based memory locations into operations on cells, taking care of byte-level aliasing between cells. We use a dynamic mapping because, due to pointer casts, the uses of the memory cannot be deduced from the static type information only. The soundness of our approach is proved in the Abstract Interpretation framework. We first construct a non-standard concrete semantics that gives a formal meaning to unclean C constructs. Then, we abstract it to derive a static analysis that is sound by construction. This makes our design modular—

it can be used with any underlying numerical domain, even a relational one such as octagons—and extensible—new abstractions based on the same concrete semantics can be designed. The abstraction is currently limited to programs without unbounded dynamic memory allocation or recursively. We deliberately left out these features as they are generally forbidden in critical software. Our work was integrated within the ASTRÉE analyzer [3] that checks for run-time errors in embedded critical C code and provides tight variable bounds, in a few hours of computation time.

Overview of the Paper. Sect. 2 motivates our work by presenting a few realistic code examples involving union types and complex pointer arithmetics; they cannot be analyzed soundly without considering byte-level aliasing. In Sect. 3, we present our solution to this problem in an intuitive way. Sect. 4 then formalizes our approach in the Abstract Interpretation framework. Sect. 5 presents preliminary experimental results obtained with the ASTRÉE analyzer. Sect. 6 presents related work. Finally, Sect. 7 discusses future work and Sect. 8 concludes.

2. The Need for a New Memory Domain

The simplest framework, when performing a value-analysis, is to consider programs with a statically known set of variables, each having a *scalar* type: real (*i.e.*, integer or floating-point) or pointer type. Such analyses can be lifted to cope with variables of *aggregate*[1] type—arrays and structures—by decomposing them into collections of independent *cells* of scalar type. Much literature has been devoted to the problems of abstracting numerical invariants, performing pointer analysis, or summarizing aggregate variables into fewer cells—to cope with large arrays or dynamic memory allocation. We are concerned here with the case where the basis hypothesis of these works fail: the memory cannot be decomposed *a priori* into a set of independent cells. This happens in a language such as C that permits very low-level accesses to the memory and the bit-representation of data.

Union Types. Union types declare fields that, unlike aggregate types, share the same memory locations. As a consequence, access paths to cells may be aliased. Consider Fig. 1 implementing message objects using a dynamic type tag `type`. In the `process` function, `m->T.type`, `m->A.type` and `m->B.type` all refer to the same cell containing an `int`. It is perfectly legal to modify the cell using one access path and read back its contents using another one [14, §6.5.2.3.5]. This kind of aliasing is quite benign as it does not prevent us from viewing the memory as a collection of distinct cells—*e.g.*, using *offsets* instead of access paths to denote cells.

A programmer may, however, disregard the value of `type`, write into `m->A.a[0]` and read back `m->B.x`, thus mixing access paths referring to (partially) overlapping memory locations of different types. Although such mixing is strongly discouraged by the C norm [14] and relies on unportable assumptions on structure layouts and value encodings, it is surprisingly often performed by programmers. Consider the variable `regs` modeling, in Fig. 4, the register state of an Intel 8086 processor. It is expected that, when modifying the word register `regs.w.ax`, its low- and hi-byte components `regs.b.ah` and `regs.b.al` are updated and can safely be read back. Due to this *byte-level* aliasing, no partition of the memory into scalar cells exists.

[1] The terms *real*, *scalar*, *aggregate* come from the C norm [14].

```
struct msgA { int type; int a[2]; };
struct msgB { int type; double x; };

union msg {
  struct { int type; } T;
  struct msgA A;
  struct msgB B;
};

void process(union msg *m) {
  switch (m->T.type) {
  case 0: {
    struct msgA* msga = &(m->A);
    int data = msga->a[0]+1;
    /* work on msga */
  }
  case 1: {
    struct msgB* msgb = &(m->B);
    /* work on msgb */
}

void read_sensor_4(unsigned* m) {
  /* put 4 bytes from sensors into m */
}

void main(void) {
  unsigned char buf[sizeof(union msg)];
  int i;
  for (i=0;i<sizeof(buf)/4;i++)
    read_sensor_4((unsigned*)buf+i);
  process((union msg*)buf);
}
```

Figure 1. Message manipulation example illustrating the use of union types.

```
void
memcopy(void* dst, void* src, unsigned sz) {
  unsigned char* s = (unsigned char*) src;
  unsigned char* d = (unsigned char*) dst;
  unsigned i;
  for (i=0;i<sz;i++) d[i] = s[i];
}

int get(unsigned char* buf) {
  struct { int *p; ··· } S;
  memcopy(&S, buf+16, sizeof(S));
  return *(S.p);
}
```

Figure 2. User-defined generic memory copy procedure.

```
void
memcopy(void* dst, void* src, unsigned sz) {
  char* s = (char*) src;
  char* d = (char*) dst;
  for (;sz>=8;sz-=8,s+=8,d+=8)
    *((double*)d) = *((double*)s);
  for (;sz!=0;sz--,s++,d++) *d = *s;
}
```

Figure 3. Alternate user-defined memory copy procedure.

Pointer Arithmetics. Pointer arithmetics encompasses array indexing. For instance, given the following declaration:

```
struct { int a[3]; int b; } U, V;
```

`*(U.a+2)` is equivalent to `U.a[2]`. But pointer arithmetics also allows escaping from an array embedded within a larger type, breaking standard out-of-bound array analyses. For instance, `*(U.a+3)` can safely be considered equivalent to `U.b` for most compilers. No assumption can generally be made, however, on the relative position of `U` and `V` in memory: `U.a[4]` is considered a run-time error and `U.a+4` points to an unspecified location outside `U`—generally not within `V`.

Pointer Casts. Pointer casts allow considering any part of the memory as having any type. Consider the `main` function in Fig. 1. It declares `buf` as an array of `unsigned char` but actually uses it both as a reference to an `unsigned int` (when calling `read_sensor_4`) and as a message of type `union msg` (when calling `process`). This achieves the same effect as a union type, except that the set of possible cell layouts is no longer embedded within the static type of the variable. It must be guessed dynamically. An extreme illustration of this problem is given by the generic memory copy functions `memcopy` of Fig. 2 (a portable, one-byte-at-a-time version) and Fig. 3 (an optimized version that copies by bunches of eight bytes, inspired from actual PowerPC software). There, the `void*` type is used to achieved polymorphism. This effectively discards all type information that would hint at the structure of the memory from `src` to `src+sz`-1. Despite this lack of typing information, we must be able to copy multi-byte cells from `src` to `dst` in a way consistent with their type. In order to treat precisely the indirect addressing `*(S.p)` at the end of the `get` function in Fig. 2, it is paramount to copy "as-is" the pointer value hidden at offset 16 in `buf`. We refer the reader to Siff et al. [21] for more examples of type casts used in real-life C programs.

3. Overview of the Analysis

In this section, we only try to present the gist of our analysis in an informal way. The next section will be devoted to its precise, mathematical definition.

3.1 Assumptions

Limitations. Our analysis computes, for each *control state*, an overapproximation of the reachable memory states, where a control state is given by a program point together with a call-stack. For the sake of simplicity, we place ourselves in the context of a fully context-sensitive analysis on code without recursive procedure nor dynamically memory allocation. Our main hypothesis is that the set $\mathcal{V}_c$ of variables whose contents define the memory state—global and local variables from all stack frames—is a static function of the control state c only. In practice, it is valid when analyzing embedded C code (where `malloc` and recursion are prohibited) with a high level of precision (requiring context-sensitivity). However, we believe that these limitations may be overcome using *summarization* techniques which are orthogonal to our purpose—*e.g.*, heap abstraction as in [20], array summarization [10], or procedure summarization [24].

Application Binary Interface. In order to achieve a high-level of description and discourage unportable practices, the C norm [14] under-specifies many parts of the language. In particular, the exact encoding of scalar types as well as the layout of fields in structures are mostly left to the implementor. However, in order to ensure the interoperability of compiled programs, libraries, and operating systems, the precise representation of types is standardized in so-called implementation-specific Application Binary Interfaces (or *ABI*) such as [1]. Although it is possible to write fully portable, ABI-neutral C code, our purpose here is the analysis of C programs that make explicit use of architecture-dependent features—such as embedded programs that need to be efficient and have a low-level access to the system. Thus, our analysis is parameterized by ABI functions, such as $\texttt{sizeof} : \mathcal{V}_c \to \mathbb{N}$ that gives the byte-size of each variable.

Input Language. We suppose that each C function has been processed into a control-flow graph where basic blocks are either assignment or guard instructions involving only side-effect free expressions. Moreover, using our knowledge of the ABI, all pointer arithmetics has been broken down to the byte-level. Except for the purpose of dereferencing, all pointers can be assumed to be pointers to `unsigned char`. All memory reads and writes are performed through pointer dereferencing. We assume that these involve only scalar types (*i.e.*, integers, floating-points, and pointers). Likewise, field selection `.` and `->` in `struct` and `union`, as well as array indexes `[]` have been converted into byte-level pointer arithmetics and dereferences of values of scalar type. As these are usual static simplifications performed by most compilers and analyzer front-ends, we do not present them in more details. Constructs that do not fit in this simplified framework (such as function pointers or assignment of compound values) will be dealt with in Sect. 4.

Numerical Analysis Parameter. Our analysis is parameterized by a standard numerical analysis. Following the Abstract Interpretation framework [5], we suppose that it is given in the form of a *numerical abstract domain*, *i.e.*, an abstract representation of invariants together with abstract *transfer functions* to mimic, in the abstract, the effect of instructions and control-flow joins. In theory, such an analysis outputs an invariant I_c on $\mathcal{V}_c$ at each program point c. However, it supposes that variables are unaliased and have real type (*i.e.*, integer or floating-point), which is not the case for $\mathcal{V}_c$. Thus, we do not use the numerical domain directly on $\mathcal{V}_c$ but on some collection $\mathcal{C}_c$ of synthetic *cells* of real type. We provide an abstraction of the memory layout that drives the numerical analysis by dynamically managing $\mathcal{C}_c$, translating instructions over $\mathcal{V}_c$ into instructions over $\mathcal{C}_c$, and taking care of byte-level aliasing between cells in $\mathcal{C}_c$.

3.2 Abstract Memory Layout

Each variable V is viewed as an unstructured sequence of $\texttt{sizeof}(V)$ contiguous bytes. Its layout in $\mathcal{C}_c$ is initially empty. It will be populated with possibly overlapping cells of scalar type as V is accessed. Abstracting a program instruction is done in three steps. First, we *enrich* the layout by adding all cells targeted by a dereference in the instruction. Secondly, we evaluate, in the numerical domain. the instruction where all dereferences have been replaced with cells. Thirdly, we *remove* all cells invalidated by alias-induced side-effects. When a layout $\mathcal{C}_c$ is changed, the corresponding cells are created or deleted in the numerical invariant I_c.

We illustrate this mechanism on the example of Fig. 4. Fig. 5 gives the abstract memory layouts $\mathcal{C}_1$ to $\mathcal{C}_7$ of the variable `regs` at program points *(1)* to *(7)*.

- The assignment `regs.w.ax = X` first creates a new cell, named **ax**, of type `uint16`, occupying offsets 0 and 1 in the variable `regs`—see the top of Fig. 5. Supposing that

```
static union {
  struct { uint8 al,ah,bl,bh,... } b;
  struct { uint16 ax,bx,... } w;
} regs;

regs.w.ax = X; (1)
if (!regs.b.ah) (2) regs.b.bl = regs.b.al; (3)
else (4) regs.b.bh = regs.b.al; (5)
(6) regs.b.al = X; (7)
```

Figure 4. Register state of an Intel 8086 processor and sample code to manipulate it. *(1)* to *(7)* represent program points of interest—see Fig. 5.

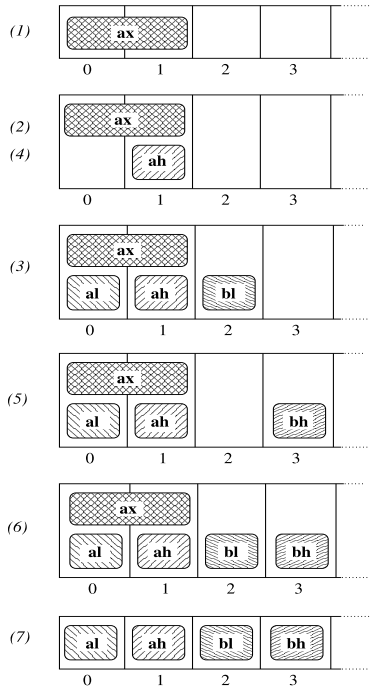

Figure 5. Memory layouts C_1 to C_7 for the variable `regs` at program points *(1)* to *(7)* when analyzing Fig. 4.

`X` corresponds to cell **X**, the assignment is then evaluated as $\mathbf{ax} \leftarrow \mathbf{X}$ in the numerical domain to yield I_1.

- Before executing the test `!regs.b.ah`, the cell **ah** of type `uint8` is created at offset 1 in `regs`. If the ABI tells us that the computer uses a little-endian byte-ordering, the cell can be initialized using the constraint $\mathbf{ah} = \mathbf{ax}/256$ on I_1. Finally, the test is executed by adding the constraint $\mathbf{ah} = 0$. When backed by a sufficiently powerful numerical domain, we may be able to infer that $\mathbf{X}/256 = \mathbf{ax}/256 = \mathbf{ah} = 0$, *i.e.*, $\mathbf{X} \in [0, 255]$, at *(2)*.
- Let us now consider the control-flow join following the conditional branches *(3)* and *(5)*. As $\mathcal{C}_3 \neq \mathcal{C}_5$, we must unify cell sets. This is done by adding missing cells: **bh** is added to $\mathcal{C}_3$ and **bl** to $\mathcal{C}_5$. As `regs` is declared `static`, its bytes are initialized to 0—according to the C norm [14]. Thus, we add the constraint $\mathbf{bh} = 0$ to I_3 and $\mathbf{bl} = 0$ to I_5. The control-flow join is then performed safely in the numerical domain, using the cell set $\mathcal{C}_6 = \mathcal{C}_3 \cup \mathcal{C}_5$.
- The last assignment, `regs.b.al = X`, can be directly evaluated as $\mathbf{al} \leftarrow \mathbf{X}$ in the numerical domain because all the cells involved exist. However, modifying **al** also modifies **ax**, a fact the numerical domain is not aware of. We correct the invariant I_7 by deleting, after the assignment, all cells that overlap the modified cells. Thus, $\mathbf{ax} \notin \mathcal{C}_7$. Note that **ax** will be back when `regs.w.ax` is accessed next and its contents will be synthesized using fresh information from the overlapping cells **ah** and **al**.
- In the presence of loops, we iterate the abstract computation until it stabilizes. Numerical domains usually use special joint-like binary operators $\triangledown^\sharp$, so called widenings [5], to accelerate fixpoint computations. As for control-flow joints, we first unify the cells layouts of the arguments, and then apply $\triangledown^\sharp$ in the numerical domain.

3.3 Pointer Abstraction

Numerical domains can only abstract directly environments over cells of real type, not pointer type. Thankfully, a pointer value can be viewed as a pair $(V, o) \in \mathcal{V}_c \times \mathbb{N}$, which represents the offset o, in bytes, from the beginning of the variable V. We abstract each component independently: one cell of integer type is allocated in the numerical domain for each pointer cell to represent its possible offsets while we maintain, in the memory layout, a map associating a set of base variables to each pointer cell. Pointer arithmetics in expressions are straightforwardly translated into integer arithmetics on offsets and then fed to the numerical domain. One benefit of this is that we are able to find relationship between pointer and integer values. For instance, when using the polyhedron domain, we can infer that `s=src+sz` holds at the end of the memory copy procedure of Fig. 3.

3.4 Intersection Semantics

As it will become clear in Sect. 4.4.3, we actually use an *intersection* semantics for overlapping cells. Suppose, for instance, that the analysis of Fig. 4 found some numerical invariant I_c with respect to the layout $\mathcal{C}_c = \{\mathbf{ah}, \mathbf{ax}\}$. Then, taking byte-level aliasing into account, `regs.b.ah` $= v$ is possible only if both **ax**'s hi-byte and **ah** can be v: I_c actually stands for $I_c \wedge (\mathbf{ah} = \mathbf{ax}/256)$. This semantics ensures that, when reading a cell's contents, it is safe to ignore overlapping cells—we simply lose some constraints, which is sound. In practice, when there already exists a cell in $\mathcal{C}_c$ with the correct type and offset—which is the most common case—we use it without looking at overlapping cells while, when it needs to be created, we use existing overlapping cells to synthesize good and safe initial values. Dually, when writing a cell's contents, we must take care to update *all* overlapping cells as they give constraints that were true before the assignment but are no longer valid. In practice, we destroy such cells, which actually delegates the update to the next time the cell is created back by a read.

Another legitimate choice would have been a *union* semantics. However, this would have made write cheap and read costly. We favored the cheap reads of the intersection semantics. Also, as we will see in Sect. 4.4.3, the intersection semantics has a very natural formalization.

For performance reasons in the numerical domain, we should avoid creating too many cells. Our scheme keeps several redundant cells per memory byte. Thankfully, redundancy is bounded by the low number of scalar types—13 as shown in Fig. 6. As cells are created lazily and destroyed often, there is few long-lived redundancy in practice. Moreover, by associating only one offset variable per pointer—and

not one for each basis variable the pointer can point to—we lose some precision but avoid a potential quadratic blow-up.

4. Formalization of the Analysis

4.1 Abstract Interpretation

We formalize our analysis in the Abstract Interpretation framework, a general theory of the approximation of program semantics introduced by Cousot and Cousot in [5]. It allows the systematic design of static analyses with various levels of precision. The gist of the method is first to design a *concrete semantics*, the most precise mathematical expression of program behaviors. This step emphases on expressibility only and generally results in a non computable semantics. Then, sound abstractions are performed and composed until a computable semantics is derived from the concrete one. This results in an abstract interpreter that can be run without user intervention, terminates on all programs, and is, by construction, sound with respects to the concrete semantics. It is, however, often incomplete. Abstractions should be carefully chosen based on the class of properties to be checked, the class of programs analyzed, and the amount of resources to be invested in the static analysis.

There exists a large library of *abstract domains* that provide ready-to-use abstract computation algorithms. Formally, given a concrete universe $\mathcal{D}$ where the concrete semantics is formalized, an abstract domain is given by:

- a set $\mathcal{D}^\sharp$ of computer-representable abstract properties,
- a concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{D})$ assigning a meaning to each abstract property,
- a computable partial order $\sqsubseteq^\sharp$ on $\mathcal{D}^\sharp$ such that γ is monotonic: $X^\sharp \sqsubseteq^\sharp Y^\sharp \implies \gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$; it models the relative precision of abstract elements and enables fixpoint abstractions through iteration schemes—potentially accelerated using special extrapolation operators $\triangledown^\sharp$,
- for each $n-$ary semantical operator $F : \mathcal{D}^n \to \mathcal{P}(\mathcal{D})$, an abstract, computable version $F^\sharp : \mathcal{D}^{\sharp n} \to \mathcal{D}^\sharp$ that is sound, *i.e.*, $\forall X_i^\sharp \in \mathcal{D}^\sharp$, $\forall X_i \in \gamma(X_i^\sharp)$, $F(X_1, \ldots, X_n) \subseteq (\gamma \circ F^\sharp)(X_1^\sharp, \ldots, X_n^\sharp)$.

We refer the reader to [5, 6] for more informations on the theory of Abstract Interpretation and its applications.

4.2 Language

We suppose that the program has been preprocessed into the simple language of Fig. 6. Each type denotes not only a set of possible values, but also their bit-representation in memory. We assume that all pointers use the same representation, and so, use a single type denoted by `ptr`. In expressions, $\mathcal{V}_c$ and $\mathcal{F}$ denote respectively the set of variables at control point c and functions. We have distinguished two kinds of assignments: assignments of expressions of scalar type, and *copy* assignments of arbitrary data structures.

4.3 Concrete Memory Domain $\mathcal{D}_M$

We now introduce a non-standard, low-level semantics $\mathcal{D}_M$ that gives a meaning to the programs of Sect. 2.

4.3.1 Concrete Memory Representation

Values. Let us denote by $\mathbb{V}_\tau$ the set of values of scalar type τ. For real types, $\mathbb{V}_\tau$ is a finite subset of $\mathbb{R}$. Pointer values range in the following set:

$$\mathbb{V}_{\texttt{ptr}} \stackrel{\text{def}}{=} \{\, (V,i) \,|\, V \in \mathcal{V}_c \cup \mathcal{F},\ 0 \le i \le \texttt{sizeof}(V) \,\} \cup \{\emptyset, \omega\}$$

int-sign	::=	`unsigned` | `signed`	
int-type	::=	`char` | `short` | `int` | `long` | `long long`	
float-type	::=	`float` | `double` | `long double`	
real-type	::=	*int-sign int-type* | *float-type*	
scalar-type	::=	*real-type* | `ptr`	
type	::=	*scalar-type*	
	|	*type*`[`n`]`	$n \in \mathbb{N}$
	|	`struct {` Id_1 : *type*, …, Id_n : *type* `}`	
	|	`union {` Id_1 : *type*, …, Id_n : *type* `}`	

expr	::=	*cst*	$cst \in \mathbb{R}$
	|	`&`V	$V \in \mathcal{V}_c \cup \mathcal{F}$
	|	$\diamond$ *expr*	$\diamond \in \{-, \sim, !\}$
	|	*expr* $\diamond$ *expr*	$\diamond \in \{+, \le, \texttt{\&}, \texttt{||}, \ldots\}$
	|	$\texttt{*}_\tau$ *expr*	$\tau \in$ *scalar-type*
	|	`(`τ`)` *expr*	$\tau \in$ *scalar-type*

inst	::=	$\texttt{*}_\tau$ *expr* $\leftarrow$ *expr*	$\tau \in$ *scalar-type*
	|	$\texttt{*}_\tau$ *expr* $\leftarrow \texttt{*}_\tau$ *expr*	$\tau \in$ *type*
	|	*expr* $== 0$?	

Figure 6. Language Syntax.

where $\emptyset$ is the `NULL` pointer while ω represents all erroneous pointer values. Valid pointers pointers are (base,offset) pairs. Following the C norm, data pointers can point one byte past the end of a variable. To treat function pointers the same way as data pointers, it is sufficient to extend `sizeof` so that $\texttt{sizeof}(V) \stackrel{\text{def}}{=} 0$ when $V \in \mathcal{F}$: valid functions pointers always have a null offset.

Memory State. We decompose the memory into a collection $\mathcal{B}$ of untyped *byte locations*:

$$\mathcal{B}(\mathcal{V}_c) \stackrel{\text{def}}{=} \{\, (V,i) \,|\, V \in \mathcal{V}_c,\ 0 \le i < \texttt{sizeof}(V) \,\}$$

The set of values a byte can hold is defined as the following set $\mathbb{V}$ of triples:

$$\mathbb{V} \stackrel{\text{def}}{=} \{(\tau, b, v) | \tau \in \textit{scalar-type}, 0 \le b < \texttt{sizeof}(\tau), v \in \mathbb{V}_\tau\}$$

where (τ, b, v) represents symbolically the $b-$th byte of the representation of the value v of scalar type τ. A concrete memory state associates a byte value to each byte location: $\mathcal{D}_M(\mathcal{V}_c) \stackrel{\text{def}}{=} \mathcal{B}(\mathcal{V}_c) \to \mathbb{V}$. Note that, in actual computers, the memory maps byte locations to numbers within $[0, 255]$. Our memory representation is sightly higher-level as it abstracts away the encoding from $\mathbb{V}$ to $[0, 255]$. For instance, the base variable of pointers is kept symbolic so that our semantics is independent from the absolute address chosen for the variables by memory allocation services.

Value Recomposition. Due to pointer casts, a sequence of bytes may be dereferenced as a value of any type. Thus, we now suppose that we are given a family of functions ϕ_τ that construct all the values of type $\tau \in$ *scalar-type* corresponding to a given byte sequence: $\phi_\tau : \mathbb{V}^{\texttt{sizeof}(\tau)} \to \mathcal{P}(\mathbb{V}_\tau)$. Note that, to allow a conservative modeling of casting, the functions may output several values. The exact definition of ϕ is highly dependent upon the ABI. We provide, in Fig. 7, an example definition valid for Intel x86 processors. It embeds useful information, such as the fact that $\emptyset$ is always represented as the integer 0, or that integers are represented using two's complement arithmetics and little endian byte ordering—it models precisely the `regs` variable in Fig. 4. However, when the value depends upon information abstracted away by our

$$\phi_\tau\langle(\tau_0,b_0,v_0),\ldots\rangle \stackrel{\text{def}}{=} \{v\} \quad \text{if } \forall k,\ v_k = v,\ \tau_k = \tau,\ b_k = k$$

$$\phi_{\texttt{unsigned char}}\langle(\tau,b,v)\rangle \stackrel{\text{def}}{=} \begin{cases} \{0\} & \text{if } \tau = \texttt{ptr} \text{ and } v = \emptyset \\ \{v/(256^b) \bmod 256\} & \text{if } \tau \in \textit{int-type} \\ [0,255] & \text{otherwise} \end{cases}$$

$$\phi_{\texttt{unsigned } t}\langle x_0,\ldots\rangle \stackrel{\text{def}}{=} \{\ \textstyle\sum_k 2^{256\times k} \times y_k \mid y_k \in \phi_{\texttt{unsigned char}}\langle x_k\rangle\ \}$$

$$\phi_{\texttt{signed } t}\langle x\rangle \stackrel{\text{def}}{=} \{\ w \mid w + 2^{\texttt{sizeof}(t)}\mathbb{Z} \cap \phi_{\texttt{unsigned } t}\langle x\rangle \neq \emptyset,\ w \in [-2^{\texttt{sizeof}(t)-1} - 1, 2^{\texttt{sizeof}(t)-1}]\ \}$$

$$\phi_{\texttt{ptr}}\langle x\rangle \stackrel{\text{def}}{=} \begin{cases} \{\emptyset\} & \text{if } \phi_{\texttt{unsigned long}}\langle x\rangle = 0 \\ \mathbb{V}_{\texttt{ptr}} & \text{otherwise} \end{cases}$$

in all other cases, $\phi_\tau\langle x\rangle \stackrel{\text{def}}{=} \mathbb{V}_\tau$

Figure 7. Value recomposition function example.

semantics (*e.g.*, non-ø pointers cannot be converted to integers without knowing the absolute address of variables) or when we are not interested in the precise behavior of a particular construction (*e.g.*, reading the binary representation of floating-point values) we use a conservative definition: $\phi_\tau\langle x\rangle = \mathbb{V}_\tau$. If need be, these cases can be refined. Dually, we may trade precision for generality—*e.g.*, drop our assumption on the byte ordering of integers.

4.3.2 Concrete Semantics

Expression Semantics. The concrete semantics $[\![e]\!] : \mathcal{D}_M(\mathcal{V}_c) \to \mathcal{P}(\mathbb{V}_\tau)$ of an expression e of type $\tau \in \textit{scalar-type}$ associates a set of values to a memory state. Most of its definition can be readily extracted from the C norm [14] and the IEEE 754-1985 norm [13]. We present here only the part related to our non-standard definition of the memory. It corresponds to the semantics of pointers and dereferences:

- $[\![\&V]\!](M) \stackrel{\text{def}}{=} \{\ (V,0)\ \}$
- $[\![e + e']\!](M) \stackrel{\text{def}}{=} \{\ (V,i+j) \mid 0 \le i+j \le \texttt{sizeof}(V),\ (V,i) \in [\![e]\!](M),\ j \in [\![e']\!](M)\ \}$
- $[\![e - e']\!](M) \stackrel{\text{def}}{=} \{\ i - j \mid (V,i) \in [\![e]\!](M),\ (V,j) \in [\![e']\!](M)\ \}$
- $[\![\texttt{(ptr)}e]\!](M) \stackrel{\text{def}}{=} \begin{cases} \{\emptyset\} & \text{if } [\![e]\!](M) \subseteq \{0,\emptyset\} \\ \mathbb{V}_{\texttt{ptr}} & \text{otherwise} \end{cases}$
- $[\![*_\tau e]\!](M) \stackrel{\text{def}}{=} \cup\ \{\ \phi_\tau\langle M(V,i),\ldots,M(V,i+\texttt{sizeof}(\tau)-1)\rangle \mid (V,i) \in [\![e]\!](M),\ i + \texttt{sizeof}(\tau) \le \texttt{sizeof}(V),\ i \equiv 0\ [\texttt{alignof}(\tau)]\ \}$

As before, the non-determinism allows a lose but sound modeling of concrete actions. Erroneous computations (such as overflows in pointer arithmetics and out-of-bound or misaligned pointers in dereferences) halt the program, and so, do not contribute to the set of accessible states.

Instruction Semantics. The semantics $\{\!| i |\!\} : \mathcal{D}_M(\mathcal{V}_c) \to \mathcal{P}(\mathcal{D}_M(\mathcal{V}_c))$ of an instruction i maps a memory state before the instruction to a set of possible memory states after the instruction. It is defined as follows:

- tests filter out environments that cannot satisfy the test:

$$\{\!| e == 0\ ?\ |\!\}(M) \stackrel{\text{def}}{=} \begin{cases} \{M\} & \text{if } 0 \in [\![e]\!](M) \\ \emptyset & \text{otherwise} \end{cases}$$

- copy assignments perform a byte-per-byte copy:

$$\{\!| *_\tau e \leftarrow *_\tau f |\!\}(M) \stackrel{\text{def}}{=} \{\ M[(V,i) \mapsto M(W,j),\ldots,(V,i+n) \mapsto M(W,j+n)] \mid (V,i) \in [\![e]\!](M),\ (W,j) \in [\![f]\!](M),\ n = \texttt{sizeof}(\tau) - 1,\ i + n < \texttt{sizeof}(V),\ j + n < \texttt{sizeof}(W)\ \}$$

- regular assignments evaluate the right-hand expression and store its byte components into the memory:

$$\{\!| *_\tau e \leftarrow f |\!\}(M) \stackrel{\text{def}}{=} \{\ M[(V,i) \mapsto (\tau,0,v),\ldots,(V,i+n) \mapsto (\tau,n,v)] \mid (V,i) \in [\![e]\!](M),\ v \in [\![f]\!](M),\ n = \texttt{sizeof}(\tau) - 1,\ i + n < \texttt{sizeof}(V)\ \}$$

Note how the value conversion ϕ due to pointer casts only occurs at memory reads, *i.e.*, in a lazy way, so as to reduce the precision loss. Most of the time, we fall in the first case of Fig. 7: we read back a byte sequence corresponding to a value v stored by a previous assignment of matching type; ϕ returns the singleton $\{v\}$ and there is no loss of precision.

When τ is scalar, $*_\tau e \leftarrow *_\tau e'$ can be considered as either type of assignments, but the copy assignment form is more precise because it avoids interpreting the memory contents via ϕ. This allows the precise modeling of the polymorphic memory copy functions of Figs. 2–3 as byte-per-byte copies.

Variable Creation and Destruction. When creating a new (zero-initialized) variable V of type τ, new byte locations initialized to the value $(\texttt{unsigned char}, 0, 0)$ are added to $\mathcal{B}$. However, deleting a variable V from a memory state M is more complex. We must not only remove some byte locations from $\mathcal{B}$, but also invalidate pointers to V in the remaining locations, which gives the following memory state:

$$(W,i) \mapsto \begin{cases} (\texttt{ptr}, j, \omega) & \text{if } M(W,i) = (\texttt{ptr}, j, (V,\cdot)) \\ M(W,i) & \text{otherwise} \end{cases}$$

4.4 Memory Abstractions

We now present computable abstractions of the concrete memory domain. We are able to retrieve, in Sect. 4.4.3, the analysis presented in Sect. 3, in a sound and formal way. We also present, in Sect. 4.4.4, a *memory equality* abstraction to improve its precision in the presence of copy assignments.

4.4.1 Scalar Value Abstraction

We suppose that we are given a numerical abstract domain $\mathcal{D}^\sharp_{\mathbb{R}}(N)$ able to abstract environments over a set N of cells with real type. That is, its concretization $\gamma_{\mathbb{R}}$ lives in $\mathcal{D}^\sharp_{\mathbb{R}}(N) \to \mathbb{R}^N$ and it features assignment and test transfer functions on expressions involving only real-valued constants, cells in N, and arithmetic operators. We refer the reader to [5, 16] for example definitions, including support for relational invariants and floating-point arithmetics.

Our first task is to add support for pointer values $\mathbb{V}_{\texttt{ptr}}$ to $\mathcal{D}^\sharp_{\mathbb{R}}(N)$. As explained in Sect. 3.3, the base component of a pointer is abstracted as a set of variables or functions while its offset is assigned a dimension in the numerical domain. Given a collection $\mathcal{C}$ of cells of scalar type, the enhanced domain $\mathcal{D}^\sharp_{\mathbb{V}}(\mathcal{C})$ is constructed as follows:

$$\mathcal{D}^\sharp_{\mathbb{V}}(\mathcal{C}) \stackrel{\text{def}}{=} \mathcal{D}^\sharp_{\mathbb{R}}(\mathcal{C}) \times (\mathcal{C}_{\texttt{ptr}} \to \mathcal{P}(\mathcal{V}_c \cup \{\omega,\emptyset\}))$$

where $\mathcal{C}_{\texttt{ptr}}$ is the subset of $\mathcal{C}$ with pointer type. A pair $(N,P) \in \mathcal{D}^\sharp_{\mathbb{V}}(\mathcal{C})$ represents the set $\gamma_{\mathbb{V}}((N,P))$ of environments $\rho : \mathcal{C} \to \cup_\tau(\mathbb{V}_\tau)$ such that, for some $\sigma \in \gamma_{\mathbb{R}}(N)$: if V

has real type, then $\rho(V) = \sigma(V)$; if V is a pointer, then either $\rho(V) \in P(V) \cap \{\omega, \emptyset\}$ or $\rho(V) \in (P(V) \setminus \{\omega, \emptyset\}) \times \{\sigma(V)\}$.

At the level of $\mathcal{D}^\sharp_\mathbb{V}$, we accept the same expressions as in $\mathcal{D}^\sharp_\mathbb{R}$ with the addition of pointer arithmetics—excluding pointer dereferencing. As pointer arithmetics has been broken down to the byte level, we can feed any instruction directly to $\mathcal{D}^\sharp_\mathbb{R}$ and obtain its effect on the offset information. The effect on pointer bases is derived by structural induction on expressions. For instance, if `p`, `q` and `i` are respectively two pointers and an integer variable, then the assignment $\{\!|\, \mathtt{q} \leftarrow \mathtt{p} + \mathtt{i} \,|\!\}^\sharp_\mathbb{V}(N, P)$ in $\mathcal{D}^\sharp_\mathbb{V}$ will return the abstract pair $(\{\!|\, \mathtt{q} \leftarrow \mathtt{p} + \mathtt{i} \,|\!\}^\sharp_\mathbb{R}(N),\ P[\mathtt{q} \mapsto P(\mathtt{p})])$ stating that `q` now points to the same base variables as `p`, and its offset is that of `p` plus `i`. The binary abstract operators—such as union $\cup^\sharp_\mathbb{V}$ and ordering $\sqsubseteq^\sharp_\mathbb{V}$—are defined point-wisely. These are quite unoriginal, and so, we do not detail them further.

4.4.2 Offset Abstraction

In practice, $\mathcal{D}^\sharp_\mathbb{R}$ is not a single numerical domain but a *reduced product* of several domains specifically chosen to fit the kinds of invariants found in an application domain—in our case, reactive control-command software, this includes plain intervals [5], relational octagons [15], and domain-specific filter domains [9]. Now that we rely on $\mathcal{D}^\sharp_\mathbb{R}$ to also abstract pointer offsets, new kinds of numerical invariants are needed and we must enrich our product. An important property to infer is pointer alignment, such as $\mathtt{p} \equiv 0$ [4] when `p` is used to access elements of byte-size 4 in an array. For this, we use the *simple congruence domain* [11, 2].

Although the combination of intervals and congruences seems sufficient in most cases, preliminary experiments suggest the need to infer invariants of the more general form $\mathtt{p} \in \sum_i [a_i, b_i] \times c_i$ to represent, *e.g.*, slices in multi-dimensional arrays. No such domain exists; its construction is left as future work. Alternate ideas include using the reduced product of linear equalities and intervals, as done by Venet [23].

4.4.3 Cell-Based Memory Abstraction $\mathcal{D}^{\sharp\mathbb{V}}_M$

Cell Universe. In order to use the value domain $\mathcal{D}^\sharp_\mathbb{V}$, we need to map memory bytes in $\mathcal{D}_M(\mathcal{V}_c)$ to cells of scalar type. Given $\rho \in \mathcal{D}_M(\mathcal{V}_c)$, for each binding $\rho(V, i) = (\tau, b, v)$, we must consider a cell of type τ at offset $i - b$ in variable V, with value $v \in \mathbb{V}_\tau$. We define the following *cell universe*:

$$\mathcal{C}_{\text{all}}(\mathcal{V}_c) \stackrel{\text{def}}{=} \{\, (V, i, \tau) \mid V \in \mathcal{V}_c,\ \tau \in \textit{scalar-type},\ i \geq 0,\ i + \mathtt{sizeof}(\tau) \leq \mathtt{sizeof}(V) \,\}$$

where (V, i, τ) corresponds to a cell of type τ starting at offset i in variable V. It models bytes at locations $(V, i + b)$ for all b in $[0, \mathtt{sizeof}(\tau) - 1]$. We will say that two cells *overlap* when the byte locations they model overlap. When extracting cells from a concrete state, we can encounter overlapping cells—*e.g.*, $(\mathtt{regs}, 0, \mathtt{uint16})$ and $(\mathtt{regs}, 1, \mathtt{uint8})$ at program point *(2)* in Fig. 4. As an abstract memory state is supposed to represent a *set* of concrete states, we must consider *a fortiori* overlapping cells to accurately model all possible memory structures.

Abstract States. An abstract memory state, in $\mathcal{D}^{\sharp\mathbb{V}}_M$, is given by a subset $\mathcal{C}$ of the cell universe, together with an abstract element in $\mathcal{D}^\sharp_\mathbb{V}(\mathcal{C})$ giving the cell contents:

$$\mathcal{D}^{\sharp\mathbb{V}}_M(\mathcal{V}_c) \stackrel{\text{def}}{=} \{\, (\mathcal{C}, X) \mid \mathcal{C} \subseteq \mathcal{C}_{\text{all}}(\mathcal{V}_c),\ X \in \mathcal{D}^\sharp_\mathbb{V}(\mathcal{C}) \,\}$$

A pair represents the following set of memory states:

$$\begin{aligned}\gamma^\mathbb{V}_M(\{(V_1, i_1, \tau_1), \ldots, (V_n, i_n, \tau_n),\ X) \stackrel{\text{def}}{=} \{\ \rho \in \mathcal{D}_M(\mathcal{V}_c) \mid\ & \forall x_1 \in \phi_{\tau_1}\langle \rho(V_1, i_1), \ldots, (V_1, i_1 + \mathtt{sizeof}(\tau_1) - 1)\rangle,\\ & \vdots\\ & \forall x_n \in \phi_{\tau_n}\langle \rho(V_n, i_n), \ldots, (V_n, i_n + \mathtt{sizeof}(\tau_n) - 1)\rangle,\\ & (x_1, \ldots, x_n) \in \gamma_\mathbb{V}(X)\ \}\end{aligned}$$

Note the universal quantifiers which mean that, when two cells from $\mathcal{C}$ overlap at a byte location (V, i), $\gamma^\mathbb{V}_M(\mathcal{C}, X)$ selects only concrete environments whose byte values at (V, i) are compatible with *both* cell values from $\gamma_\mathbb{V}(X)$. Hence the term *intersection semantics* used in Sect. 3.4. Moreover, when a byte location is not covered by any cell in $\mathcal{C}$, it can take any value in the concrete world.

Cell Realization. It would be conceptually simpler to always consider $\mathcal{C} = \mathcal{C}_{\text{all}}$, but quite costly as the time and memory complexity of $\mathcal{D}^\sharp_\mathbb{V}$ depends directly on the size of $\mathcal{C}$. Thus, $\mathcal{C}$ is chosen dynamically. As $\gamma^\mathbb{V}_M$ has universal quantifiers, it is always safe to remove any cell c from $\mathcal{C}$: $\gamma^\mathbb{V}_M(\mathcal{C}, X) \subseteq \gamma^\mathbb{V}_M(\mathcal{C} \setminus \{c\}, X_{|_{\mathcal{C}\setminus\{c\}}})$. Adding a new cell c is more complex: we must initialize its value according to existing cells overlapping c so as not to forget any concrete state. We call this operation *cell realization*. First, the cell is created and initialized to $\mathbb{V}_\tau$, which is sound. Then, the value is refined by scanning the set of overlapping cells for certain patterns and applying tests transfer functions in $\mathcal{D}^\sharp_\mathcal{V}$ accordingly. For instance, when trying to add the cell **ah** in the cell set $\mathcal{C}_1$ of Fig. 5, one finds the overlapping cell **ax**. According to the $\phi_{\mathtt{unsigned\ char}}$ function of Fig. 7, we can apply the transfer function $\{\!|\, \mathbf{ax}/256 - \mathbf{ah} == 0\,? \,|\!\}^\sharp_\mathbb{R}$. Note that, if $\mathcal{D}^\sharp_\mathbb{R}$ contains relational domains, the relationship between the realized and the overlapping cells will be kept. For instance, if $\mathcal{D}^\sharp_\mathbb{R}$ is able to represent the invariant $\mathbf{ah} = \mathbf{ax}/256$, then ,whenever we learn something new on the value of one cell, it will be immediately reflected on the other one.

Abstract Operators. Assignments and tests are transformed by replacing dereferences with cell sets, and then fed to the underlying value domain. Given a sub-expression $*_\tau\, e$, where e is dereference-free, e is first evaluated in $\mathcal{D}^\sharp_\mathbb{V}$ which returns the set S of byte locations it can point to. All cells $\mathcal{C}' = \{\ (V, i, \tau),\ \mid (V, i) \in S\ \}$ are then realized in the current abstract state—if not already there. The resolution continues with the enriched abstract state for the expression where $*_\tau\, e$ has been replaced with the cell set $\mathcal{C}'$. Tests can be directly executed in $\mathcal{D}^\sharp_\mathbb{V}$ on the resulting expressions. Assignments are a little more complex because they involve memory writes. Given an assigned cell c, we first realize c, then execute the assignment in $\mathcal{D}^\sharp_\mathbb{V}$, and finally remove *all* cells overlapping c. Note that a dereference may resolve in more than one cell, $|\mathcal{C}'| > 1$, which results in *weak updates* in $\mathcal{D}^\sharp_\mathbb{V}$. We now define the abstraction $\circ^{\sharp\mathbb{V}}_M$ of a binary operator $\circ$. Given the states $S_1 = (\mathcal{C}_1, X_1)$ and $S_2 = (\mathcal{C}_2, X_2)$, we first unify the cell sets using realization to obtain two states $S'_1 = (\mathcal{C}_1 \cup \mathcal{C}_2, X'_1)$ and $S'_2 = (\mathcal{C}_1 \cup \mathcal{C}_2, X'_2)$. We then apply the binary operator on the underlying value domain and get $S_1 \circ^{\sharp\mathbb{V}}_M S_2 = (\mathcal{C}_1 \cup \mathcal{C}_2, X'_1 \circ^\sharp_\mathbb{V} X'_2)$. This is sound with respect to overlapping cells. However, because overlapping cells have an intersection semantics, we may lose some precision on $\cup^{\sharp\mathbb{V}}_M$—informally, we over-approximate $(a \cap b) \cup (c \cap d)$ as $(a \cup c) \cap (b \cup d)$. The widening $\nabla^{\sharp\mathbb{V}}_M$ stabilizes invariants by first stabilizing the cell set—which is an increasing subset of the finite set $\mathcal{C}_{\mathtt{all}}$—and then relies on the underlying

widening $\nabla^\sharp_{\mathbb{V}}$. The abstract order $\sqsubseteq^{\sharp\mathbb{V}}_M$ is defined as $\sqsubseteq^\sharp_{\mathbb{V}}$ after cell sets have been unified to $\mathcal{C}_1 \cup \mathcal{C}_2$.

There are strong similarities between the abstract cell realization and the concrete value recomposition ϕ. Both are used, in a lazy way, to reconstruct information when the type of a dereference mismatches that of the currently stored value. Both are defined according to an ABI and the level of modeling required by the user. Both may result in a loss of precision. Thus, once a cell is realized, we try to keep it around as long as possible (*i.e.*, until it is invalidated by a memory write).

4.4.4 Memory Equality Predicate Domain $\mathcal{D}^{\sharp\mathrm{Eq}}_M$

When analyzing generic memory copy functions, $\mathcal{D}^{\sharp\mathbb{V}}_M$ sometimes lacks the required precision. Consider, for instance, calling the function `memcopy(&a,&b,4)` from Fig. 2, `a` and `b` being 4-byte integers. Although it is equivalent to the plain assignment `a=b`, it is carried-out one byte at a time. $\mathcal{D}^{\sharp\mathbb{V}}_M$ will first realize individual bytes in `b` as `char` cells, copy them into `a` and, the first time `a` is read, realize back the four `char` cells as a single integer cell. Because each realization may result in some loss of precision, the inferred value set for the cell $(\mathtt{a}, 0, \mathtt{int})$ may be much larger than that of $(\mathtt{b}, 0, \mathtt{int})$.

In order to solve this problem, we introduce a specific abstraction $\mathcal{D}^{\sharp\mathrm{Eq}}_M$ of $\mathcal{D}_M$ that tracks equalities between byte values in a symbolic way:

$$\mathcal{D}^{\sharp\mathrm{Eq}}_M(\mathcal{V}_c) \stackrel{\text{def}}{=} \mathcal{V}_c \to ((\mathbb{N} \times \mathcal{V}_c \times \mathbb{N} \times \mathbb{N}) \cup \{\top^{\sharp\mathrm{Eq}}\})$$

where a binding $V \mapsto (s, W, d, l)$ means that the l bytes starting at location (V, s) are equal to those starting at location (W, d), while $\top^{\sharp\mathrm{Eq}}$ means "no information:"

$$\gamma^{\mathrm{Eq}}_M(\epsilon) \stackrel{\text{def}}{=} \{\ \rho \in \mathcal{D}_M(\mathcal{V}_c) \mid \forall V \in \mathcal{V}_c,\ \epsilon(V) = (s, W, d, l) \implies \forall 0 \le i < l,\ \rho(V, s+i) = \rho(W, d+i)\ \}$$

Note that only one predicate is kept per variable, and the parameters (s, W, d, l) are bound to concrete values. This ensures efficient transfer functions but requires memory copy loops to be fully unrolled. (We could benefit from more complex predicate abstraction schemes to overcome this restriction—*e.g.*, use [4] to keep (s, d, l) symbolic and relate their value in $\mathcal{D}^\sharp_{\mathbb{R}}(N)$. This was not required in our experience as the codes we analyze only copy small structures.)

Among instructions, only copy assignments are treated precisely: tests are safely ignored while other assignments are dealt with by removing bindings involving the destination—*i.e.*, setting them to $\top^{\sharp\mathrm{Eq}}$. Suppose that $\epsilon(V) = (s, W, d, l)$ and we copy l' bytes from (V, s') to (W', d'); several cases arise. When $W = W'$, $s - d = s' - d'$ and $s' \in [s, s+l]$, we copy bytes at the end of equal zones. We thus grow the zones by setting $\epsilon(V) = (s, W, d, \max(l, l' - s' + s))$. The case is similar when bytes are copied at the start of zones: $W = W'$, $s - d = s' - d'$ and $s \in [s', s'+l']$. In all other cases, the former binding is useless and we replace it by a new one $\epsilon(V) = (s', W', d', l')$. As W' is modified, we must also, in all cases, remove any other binding involving W'. We say that $\epsilon_1 \sqsubseteq^{\sharp\mathrm{Eq}}_M \epsilon_2$ whenever, for every V, either $\epsilon_2(V) = \top^{\sharp\mathrm{Eq}}$ or $\epsilon_1(V)$ corresponds to a sub-range of $\epsilon_2(V)$. This order has a least upper bound, which serves to define the abstract union, but no greatest lower bound. As $\mathcal{D}^{\sharp\mathrm{Eq}}_M$ has a finite height, no widening is necessary to help the iterates converge.

We perform a partially reduced product between $\mathcal{D}^{\sharp\mathbb{V}}_M$ and $\mathcal{D}^{\sharp\mathrm{Eq}}_M$. All abstract operations are performed in parallel. In addition, we propagate information from $\mathcal{D}^{\sharp\mathrm{Eq}}_M$ to $\mathcal{D}^{\sharp\mathbb{V}}_M$ after each copy assignment. For each cell (V, o, τ), if we just discovered that $\epsilon(V) = (s, W, d, l)$ and $[o, o + \mathtt{sizeof}(\tau)] \subseteq [s, s+l]$, then we realize the cell $(W, o-s+d, \tau)$ and perform the assignment $*_\tau(\&W + o - s + d) \leftarrow *_\tau(\&V + o)$ in $\mathcal{D}^{\sharp\mathbb{V}}_M$. In our example, `memcopy(&a,&b,4)`, we would generate the assignment `a`$\leftarrow$`b` just after copying the 4$-$th byte. Thus, the value for the cell $(\mathtt{a}, 0, \mathtt{int})$ is precisely that of $(\mathtt{b}, 0, \mathtt{int})$ and no longer need to be realized from the value of `char` cells.

	old domain		new domain		both
lines	time	mem.	time	mem.	alarms
9 500	82s	0.2GB	99s	0.2GB	1
70 000	62m	1.0GB	63m	1.1GB	0
226 000	4h57	1.6GB	4h42	1.7GB	1
400 000	11h04	3.0GB	11h46	3.2GB	0

Figure 8. Regression tests for Astrée.

5. Experiments

5.1 Presentation of the Astrée Analyzer

Scope. The goal of Astrée is to detect statically all run-time errors in embedded reactive software written in C. Run-time errors include integer and floating-point arithmetics overflows, divisions by zero and array out-of-bound accesses. To achieve this goal, Astrée performs an abstract reachability analysis and computes the set of values each variable can take, considering all program executions in all possible environments. To be efficient, it performs many sound but incomplete abstractions. As a consequence, it always finds *all* run-time errors but may report spurious alarms. Its abstractions are tuned towards specific classes of programs in order to achieve *zero false alarms* in practice, within reasonable time. Indeed, [3] reports its success in proving automatically the absence of run-time errors in real industrial code of several hundred thousand lines, in a few hours.

Architecture. Astrée has a modular architecture. It relies on a product of several numerical domains, which can be plug in and out. They exchange information via configurable reductions. It also features a parameterisable abstract iterator tailored for flow- and context-sensitive analysis, and trace partitioning to achieve partial path-sensitivity. In previous work [3], it has been specialised towards the analysis of embedded avionics software by incorporating adapted iterations strategies and numerical domains (such as relational octagons [15] and domain-specific filter domains [9]). However, its memory model was limited to simple well-structured data only, which was sufficient at that time. In order to analyze new code featuring union types and pointer casts, we replaced it with our new memory abstractions. Thanks to the modular construction of Astrée and its modular proof of correctness, most parts were not tied to the old memory abstraction and could be reused (in particular, all numerical and partitioning domains, as well as the iterator).

5.2 Preliminary Experimental Results

We have run three kinds of experiments: small case studies, regression tests and preliminary analyses of new real-life software. They all ran on a 64-bit AMD Opteron 250 (2.4GHz) workstation, using one processor. The analyzed programs do not feature recursion, dynamic memory allocation, nor multi-threading. Moreover, they are self-contained: they do not call precompiled library routines, and the external environment is modeled using *volatile* variables.

source	lines	time	memory	alarms
end-user 1	35 000	12m	212MB	22
	46 000	16m	271MB	84
end-user 2	92 000	3h17	3.2GB	71
	184 000	4h55	1.1GB	36

Figure 9. Four newly analyzed codes, from two end-users.

Firstly, we tested the relevance of our domains to the specific problems of union types and pointer casts discussed in Sect. 2. We produced and were given by end-users several constructed programs of a hundred lines, in the spirit of Figs. 1–4. We were able to prove the absence of run-time errors of all case studies in a fraction of second.

Secondly, we re-analyzed the pointer- and union-free industrial embedded critical code successfully analyzed by Astrée in previous work [3]. Fig. 8 compares the performance of the old and new memory domains. We see that the memory peak and time consumption are only slightly increased, in the worse case, and we find the same alarms. Note that, as Astrée uses incomplete methods—such as partially reduced products and convergence acceleration—there is no theoretical guarantee that our new memory semantics always gives more precise results than the former one, even though it is more expressive. Hence, the importance of asserting experimentally non-regression in terms of precision.

Thirdly, we analyzed four new industrial critical embedded software featuring unions and complex pointer manipulations. Such codes could not be considered before in Astrée because of its limited legacy memory domain. The analyses results are shown in Fig. 9. These results are preliminary in the sense that we have not yet investigated the causes of all alarms: they may be due to analyzer inaccuracies, but also to real errors or too conservative assumptions on the environment. The results are encouraging: they correspond to the preliminary results obtained on the codes of Fig. 8 before domain-specific numerical domains and iteration strategies were incorporated in Astrée to achieve zero alarm [3].

6. Related Work

Several dialects of C, such as CCured [17], have been proposed to prevent error-prone uses of unions and pointers. The value analysis of such dialects, with their cleaner memory model, would be easier than the full C. Unfortunately, their strengthened type systems would reject constructs found legitimate by end-users and force them to rewrite their software. For now, we (analysis designers) should adapt our analyses to the programming features they currently use.

There exists a very large body of work concerning pointer analyses for C—we refer the reader to the very good survey by Hind [12]. Unfortunately, they cannot serve our purpose. All field-insensitive methods natively support union and pointer casts—they are considered "no-op." However, in order to find precise bounds on values stored into and then fetched from memory, we absolutely require field sensitivity. Very few field-sensitive analyses support unions or casts. Most of them—*e.g.*, the recent work of Whaley and Lam [24]—assume a memory model *à la* Java, where the memory can be *a priori* partitioned into cells of unchanging type. As a middle-ground, Yong et al. [27] propose to collapse fields upon detecting accesses through pointers whose type mismatches the declared type of the fields. This is not sufficient to treat precisely union types—Fig. 1—or polymorphism—Fig. 2. Also, flow-insensitive analyses (such as the union- and cast-aware analysis by Steensgaard [22]) which are well-suited for program optimization and understanding, would not perform precise-enough for value analysis. Indeed, they tend to produce large points-to sets—especially given that we are field-sensitive—which results in weak updates and precision losses in the numerical domains. When it comes to program correctness, we are ready to use much more costly abstractions: each instruction proved correct automatically saves the user an expensive manual proof.

Instead of relying on the structure of C types, we chose to represent the memory as flat sequences of bytes. This allows shifting to a representation of pointers as pairs: a symbolic base and a numeric offset. It is a common practice—it is used, for instance, by Wilson and Lam in [25]. This also suggests combining the pointer and value analyses into a single one—offsets being treated as integer variables. There is experimental proof [18] that this is more precise than a pointer analysis followed by a value analysis. Some authors rely on non-relational abstractions of offsets—*e.g.*, a reduced product of intervals and congruences [2], or intervals together byte-size factors [26]. Others, such as [23, 19] or ourself, permit more precise, relational offset abstractions.

We stress on the fact that using an offset-based pointer representation solves, by itself, the problem of points-to analysis in the presence of union types and casts, but it does not solve the problem of analyzing precisely the *contents* of the memory such offset-based pointers point to. Several kinds of solution have been used to avoid treating this second problem. A first one is to perform the field-sensitive points-to and value analysis of only a part of the memory that is never accessed through casts—*e.g.*, the *surface structure* of [23]—while the rest is only checked for in-bound accesses. A second one is to fix one memory layout—using, *e.g.*, the declared variable types or some pointer alignment constraints—and conservatively assume that mismatching dereferences result in any value [2]. A less conservative solution, proposed by Wilson and Lam [25], is to consider that a dereference can output the value of any overlapping cell. We are more precise and more general because we allow value recomposition form individual bytes of partially overlapping cells and take into account the bit-representation of types. In particular, unlike previous work, we can analyze precisely the indirect dereferencing following the memory copy of Fig. 2. Moreover, while [25] often resolves a dereference into several overlapping cells, even when the target of the dereference is precisely known, we manage to select a single cell most of the time. This reduces the possibility of weak updates and improves the analysis precision, especially when using relational numerical domains. To our knowledge, our method is the first one that allows discovering precise *relational* invariants in the presence of union types and pointer casts.

Finally, note that most articles—[23] being a notable exception—directly leap from a memory model informally described in English to the formal description of a static analysis. Following the Abstract Interpretation framework, we give a full mathematical description of the memory model before presenting computable abstractions proved correct with respect to the model.

7. Future Work

A first goal is to reduce the number of alarms in the newly analyzed codes of Fig. 9. In the best scenario, most inaccuracies will be solved by tweaking already existing parameters—such as the level of path sensitivity or domain relationality.

However, we will probably also need to add new numerical domains in the reduced product $\mathcal{D}^\sharp_\mathbb{R}$, as it was necessary in order to achieve the proofs of absence of run-time errors in [3]. We plan to investigate particularly the numerical domains required to abstract pointer offsets precisely, as it is a new requirement of our memory abstractions. Finally, by iterating the analyzer refinement process over other codes involving unions and pointers, we hope to provide a library of abstractions that, in practice, is sufficient to analyze a large class of embedded C programs.

Further goals include incorporating domains for heap-allocated objects—*e.g.*, related to predicate-based summarization as proposed by Sagiv et al. [20, 10]. We also wish to include other memory abstractions within our framework, for instance, the string abstraction by Dor et al. [7] as well as generalizations of $\mathcal{D}^{\sharp\mathrm{Eq}}_M$ using predicate abstractions parameterized by numerical domains *à la* Cousot [4].

8. Conclusion

In this article, we proposed new techniques to perform the precise value analysis of C programs with pointers and union types. We first gave a precise meaning to such programs by defining a concrete memory semantics, parameterized by an Application Binary Interface. We then proposed two computable abstractions: a value abstraction, parameterized by the choice of a numerical abstract domain, and an equality predicate abstraction, able to precisely deal with polymorphic memory copies. The combined abstractions have been implemented within the Astrée parametric static analyzer that checks for run-time errors in embedded critical C software. Preliminary experimental results are encouraging: while not sacrificing the precision and efficiency of Astrée on legacy analyses—in particular, the proof of absence of run-time errors for some large industrial codes in a few hours of computation time—we greatly enlarge the class of analyzable programs. Currently, small test cases containing pointers and unions have been proved correct while there are still a few dozens alarms on real-life industrial examples. We are confident that these results will be improved in the future by refining the analyzer.

Acknowledgments

We would like to thanks the whole Astrée team [3], as well as the anonymous referees for their insightful comments.

References

[1] AT&T and The Santa Cruz Operation Inc. System V application binary interface, 1997.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC 2004*, number 2985 in LNCS, pages 5–23. Springer, 2004.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM PLDI'03*, volume 548030, pages 196–207. ACM Press, 2003.

[4] P. Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772, pages 243–268. Springer, 2003.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*, pages 238–252. ACM Press, 1977.

[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[7] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *SAS'01*, volume 2126 of *LNCS*. Springer, 2001.

[8] J.-L. Lions et al. ARIANE 5, flight 501 failure, report by the inquiry board, 1996.

[9] J. Feret. Static analysis of digital filters. In *ESOP'04*, volume 2986 of *LNCS*. Springer, 2004.

[10] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *TACAS 2004*, LNCS, pages 512–529. Springer, 2004.

[11] P. Granger. Static analysis of arithmetical congruences. In *International Journal of Computer Mathematics*, volume 30, pages 165–190, 1989.

[12] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[13] IEEE Computer Society. IEEE standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std. 745-1985, 1985.

[14] International Organisation for Standardization. Programming languages – C. Technical report, ISO/IEC 9899:1999, 1999.

[15] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, 2001.

[16] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[17] G. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL'02*, pages 128–139. ACM Press, 2002.

[18] A. Pioli and M. Hind. Combining interprocedural pointer analysis and conditional constant propagation. Technical Report 99-103, IBM, 1999.

[19] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI'00*, pages 182–195. ACM Press, 2000.

[20] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3), 2002.

[21] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *ESEC/FSE'99*, pages 180–198. Springer.

[22] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *CC'96*, volume 1060 of *LNCS*, pages 136–150. Springer, 1996.

[23] A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *SAS'04*, number 3148 in LNCS, pages 149–164. Springer, 2004.

[24] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *SAS'02*, volume 2477, pages 180–195. Springer.

[25] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI'95*, pages 1–12. ACM Press, 1995.

[26] S. Yong and S. Horwitz. Pointer-range analysis. In *SAS'04*, number 3148 in LNCS, pages 133–148. Springer, 2004.

[27] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *PLDI'99*, pages 91–103. ACM Press, 1999.